SAND2019-0279PE

Runtime Polymorphism in Kokkos Applications

PRESENTED BY

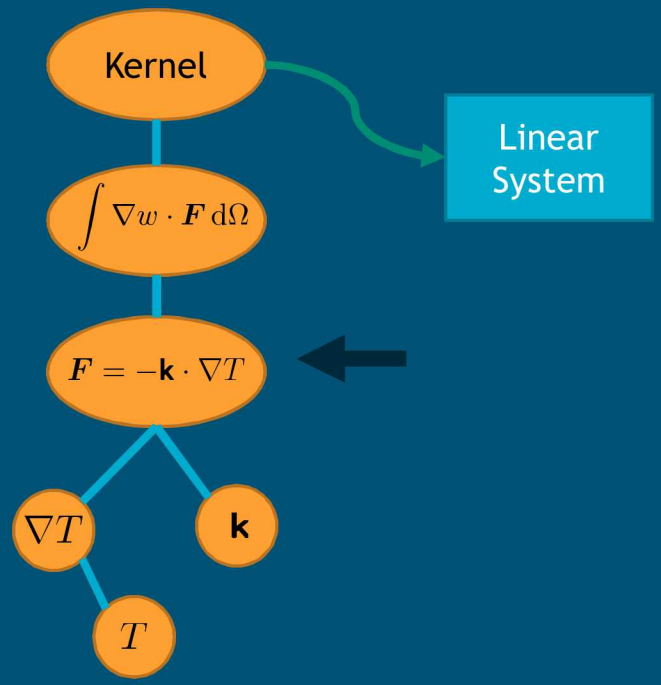
Victor Brunini

Jonathan Clausen, Mark Hoemmen, Alec Kucala,
Christian Trott, Micah Howard

Motivation

Aria

Expression graph implementation to enable flexible fully-implicit coupling of different PDEs and material models selected at runtime



SPARC

- Goal: solve aerodynamics problems for Sandia (transonic and hypersonic) on 'leadership' class supercomputers
- Solves compressible Navier-Stokes equations
- Perfect and reacting gas models
- Laminar and RANS turbulence models -> hybrid RANS-LES
- Primary discretization is cell-centered finite volume
- Research on high-order finite difference and discontinuous Galerkin discretizations
- Structured and unstructured grids

Options for runtime dispatch on GPU

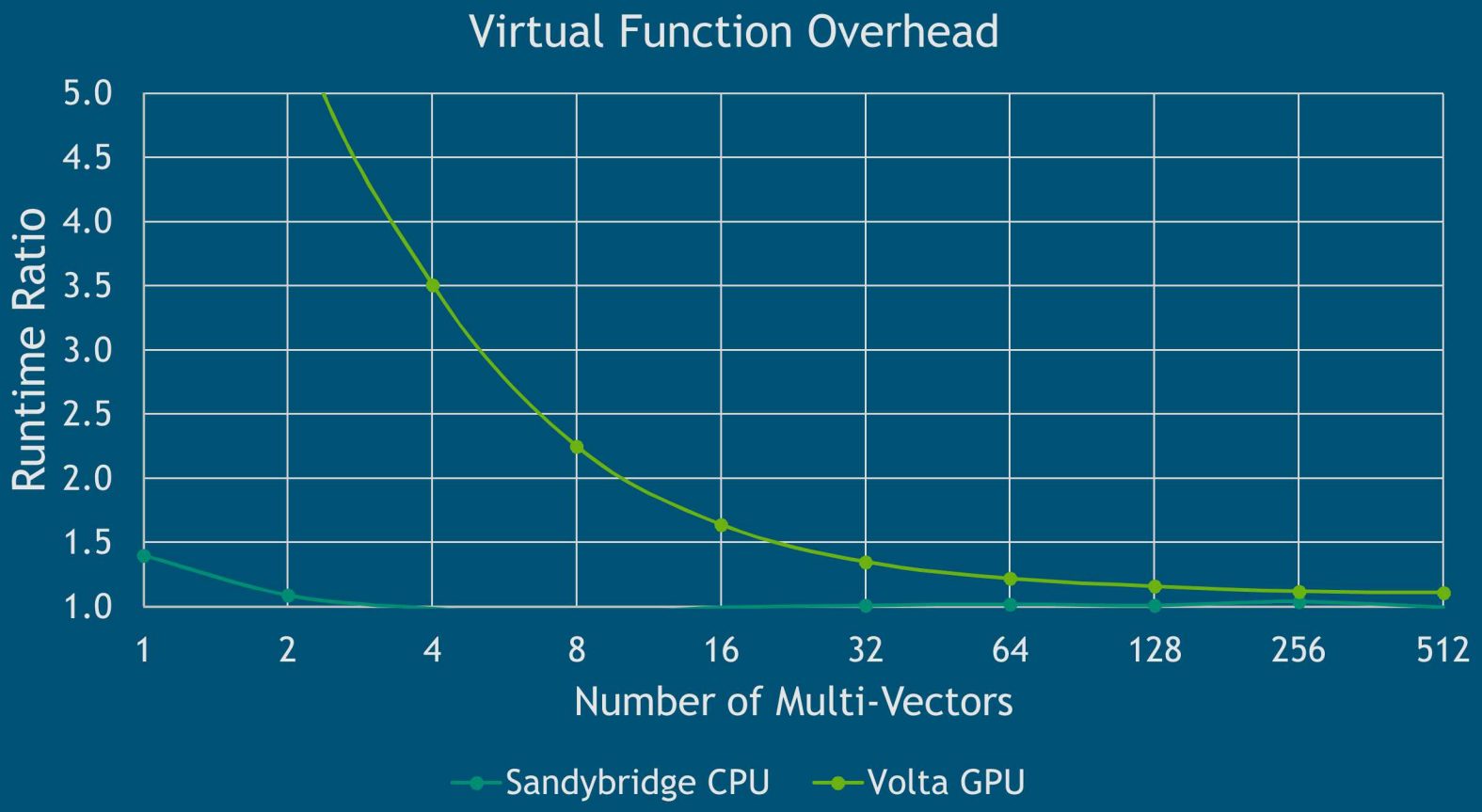
1. Use C++ templates to generate different kernels for each possible combination of element type, material models, PDE's to solve and select the appropriate one at runtime
   - Best possible performance
   - Very negative impact on compile times for even modest numbers of combinations – O(Na * Nb * Nc * …)
   - May not be viable depending on number of possible combinations to support
2. Use standard C++ virtual functions + inheritance as on the CPU
   - Requires care to ensure that vtable points to appropriate host or device functions depending on where the virtual call is going to be made
   - Can this achieve acceptable performance on the GPU?

Virtual Function Performance on GPU

Benchmark Problem: AXPY over multi-vector

```
void run_inline(const int n, const double a,
  const View<double **> & x, const View<double **> & y)
{
  const int extent = x.extent_int(1);
  parallel_for(n, KOKKOS_LAMBDA(const int i) {
      for(int j=0; j < extent; ++j) {
        y(i, j) += a * x(i, j);
      }});
}
void run_virtual(const int n, const double a,
  const View<double **> & x, const View<double **> & y)
{
  VirtualInterface * impl = get_implementation();
  parallel_for(n, KOKKOS_LAMBDA(const int i)
    { impl->compute(i, a, x, y); });
}
```

Virtual Function Performance on GPU



Above 32-64 FMA operations per call overhead on GPU levels off around 10%

Virtual Functions in Kokkos Parallel Regions

```
class VirtualInterface
{
public:
  KOKKOS_FUNCTION virtual void compute(const int i) = 0;
};

class Implementation : public VirtualInterface
{
public:
  KOKKOS_FUNCTION void compute(const int i) override;
};
```

Virtual Functions in Kokkos Parallel Regions

```
// How do we construct Implementation to call parallel_dispatch in
// a GPU build?
void parallel_dispatch(VirtualInterface * v, const int n)
{
  parallel_for(n, KOKKOS_LAMBDA(const int i) { v->compute(i); });
}
```

Virtual Functions in Kokkos Parallel Regions

```
// How do we construct Implementation to call parallel_dispatch in
// a GPU build?
void parallel_dispatch(VirtualInterface * v, const int n)
{
  parallel_for(n, KOKKOS_LAMBDA(const int i) { v->compute(i); });
}

void f()
{
  Implementation i;
  parallel_dispatch(&i, 1000);
}
```

Virtual Functions in Kokkos Parallel Regions

```
// How do we construct Implementation to call parallel_dispatch in
// a GPU build?
void parallel_dispatch(VirtualInterface * v, const int n)
{
  parallel_for(n, KOKKOS_LAMBDA(const int i) { v->compute(i); });
}

void f()
{
  Implementation i;
  parallel_dispatch(i, 1000);
}
```

Illegal Instruction Error

Virtual Functions in Kokkos Parallel Regions

```
VirtualInterface * get_implementation()
{
  Implementation * p = static_cast<Implementation *>(
     kokkos_malloc<>(sizeof(Implementation)));
  parallel_for(1,
      KOKKOS_LAMBDA(const int i) { new (p) Implementation(); });
  fence();
  return p;
}

void f()
{
  auto i = get_implementation();
  parallel_dispatch(i, 1000);
}
```

Virtual Functions in Kokkos Parallel Regions

```
VirtualInterface * get_implementation()
{
  Implementation * p = static_cast<Implementation *>(
     kokkos_malloc<>(sizeof(Implementation)));
  parallel_for(1,
      KOKKOS_LAMBDA(const int i) { new (p) Implementation(); });
  fence();
  return p;
}

void f()
{
  auto i = get_implementation();
  parallel_dispatch(i, 1000);
}
```

Success

Virtual Functions in Kokkos Parallel Regions

```
VirtualInterface * get_implementation()
{
  Implementation * p = static_cast<Implementation *>(
     kokkos_malloc<>(sizeof(Implementation)));
  parallel_for(1,
      KOKKOS_LAMBDA(const int i) { new (p) Implementation(); });
  fence();
  return p;
}

void f()
{
  auto i = get_implementation();
  parallel_dispatch(i, 1000);
  i->compute(0);
}
```

Virtual Functions in Kokkos Parallel Regions

```
VirtualInterface * get_implementation()
{
  Implementation * p = static_cast<Implementation *>(
     kokkos_malloc<>(sizeof(Implementation)));
  parallel_for(1,
      KOKKOS_LAMBDA(const int i) { new (p) Implementation(); });
  fence();
  return p;
}

void f()
{
  auto i = get_implementation();
  parallel_dispatch(i, 1000);
  i->compute(0);
}
```

Illegal Instruction Error

More Generically

```
template <typename Device, typename Derived>
Derived * copy_for_device(const Derived & to_copy)
{
  auto * p = static_cast<Derived *>(
     kokkos_malloc<typename Device::memory_space>(sizeof(Derived)));
  parallel_for(RangePolicy<typename Device::execution_space>(0, 1),
      KOKKOS_LAMBDA(const int i) { new (p) Derived(to_copy); });
  fence();
  return p;
}
```

More Generically

```
template <typename Device, typename Derived>
Derived * copy_for_device(const Derived & to_copy)
{
  auto * p = static_cast<Derived *>(
     kokkos_malloc<typename Device::memory_space>(sizeof(Derived)));
  parallel_for(RangePolicy<typename Device::execution_space>(0, 1),
      KOKKOS_LAMBDA(const int i) { new (p) Derived(to_copy); });
  fence();
  return p;
}
```

But how to free this object?

More Generically

```
template <typename Device, typename Derived>
unique_ptr<Derived> copy_for_device(const Derived & to_copy)
{
  auto * p = static_cast<Derived *>(
     kokkos_malloc<typename  Device::memory_space>(sizeof(Derived)));
  parallel_for(RangePolicy<typename Device::execution_space>(0, 1),
      KOKKOS_LAMBDA(const int i) { new (p) Derived(to_copy); });
  fence();
  return unique_ptr<Derived>(p);
}
```

More Generically

```
template <typename Device, typename Derived>
unique_ptr<Derived> copy_for_device(const Derived & to_copy)
{
  auto * p = static_cast<Derived *>(
     kokkos_malloc<typename Device::memory_space>(sizeof(Derived)));
  parallel_for(RangePolicy<typename Device::execution_space>(0, 1),
      KOKKOS_LAMBDA(const int i) { new (p) Derived(to_copy); });
  fence();
  return unique_ptr<Derived>(p);
}
```

Invalid Delete on Destruction

More Generically

```
template <typename Device>
struct DeviceDeleter
{
  template <typename T> void operator()(T * ptr)
  {
    parallel_for(RangePolicy<typename Device::execution_space>(0, 1),
      KOKKOS_LAMBDA(const int i) { ptr->~T(); });
    kokkos_free<typename Device::memory_space>(ptr);
  }
};
```

- Alternate implementation: DeviceDeleter stores an enum for device vs host instead of being a template

More Generically

```
template <typename Device, typename Derived>
unique_ptr<Derived, DeviceDeleter<Device>> copy_for_device(const Derived &
to_copy)
{
  auto * p = static_cast<Derived *>(
     kokkos_malloc<typename Device::memory_space>(sizeof(Derived)));
  parallel_for(RangePolicy<typename Device::execution_space>(0, 1),
      KOKKOS_LAMBDA(const int i) { new (p) Derived(to_copy); });
  fence();
  return unique_ptr<Derived, DeviceDeleter<Device>>(p);
}
```

View Ownership & Device Objects

```
class ImplWithView : public VirtualInterface
{
public:
  KOKKOS_FUNCTION ~ImplWithView() = default;
  KOKKOS_FUNCTION void compute(const int i) override
  { v(i) += 1; }

  KOKKOS_FUNCTION ImplWithView(const ImplWithView &) = default;
  View<double *> v;
};

void f()
{
  auto i = copy_for_device<DefaultExecutionSpace, ImplWithView>(
    ImplWithView{View<double *>("my_view", 1000)});
  parallel_dispatch(i.get(), 1000);
}
```

View Ownership & Device Objects

```
class ImplWithView : public VirtualInterface
{
public:
  KOKKOS_FUNCTION ~ImplWithView() = default;
  KOKKOS_FUNCTION void compute(const int i) override
  { v(i) += 1; }

  KOKKOS_FUNCTION ImplWithView(const ImplWithView &) = default;
  View<double *> v;
};

void f()
{
  auto i = copy_for_device<DefaultExecutionSpace, ImplWithView>(
    ImplWithView{View<double *>("my_view", 1000)});
  parallel_dispatch(i.get(), 1000);
}
```

View Allocation Already Freed!

Ref counting is host-only

View Ownership & Device Objects

```
template <typename T, typename Device>
class DeviceUniquePtr
{
public:
  // Interface mimics unique_ptr
private:
  unique_ptr<Derived, DeviceDeleter<Device>> device_ptr;
  unique_ptr<Derived> host_ptr;
};
template <typename Device, typename Derived>
DeviceUniquePtr<Derived, DeviceDeleter<Device>> copy_for_device(const Derived
  & to_copy)
{
  auto * p = static_cast<Derived *>(
     kokkos_malloc<typename Device::memory_space>(sizeof(Derived)));
  parallel_for(RangePolicy<typename Device::execution_space>(0, 1),
      KOKKOS_LAMBDA(const int i) { new (p) Derived(to_copy); });
  fence();
  return DeviceUniquePtr<Derived, DeviceDeleter<Device>>(p);
}
```

Additional Lessons Learned

Must enable relocatable device code

- nvlink is very picky about libraries appearing on the link line multiple times, nvcc_wrapper helps
- Kernels that call non-inline functions must assume maximum register usage, -maxrregcount compiler flag can be a useful optimization tool when occupancy-limited

Short kernel launches & Cuda allocations are expensive

- Constructing many polymorphic device objects per-timestep is problematic, make persistent if possible

PERFORMANCE ANALYSIS: INTRODUCTION

Performance analysis on 3 primary architecture types:

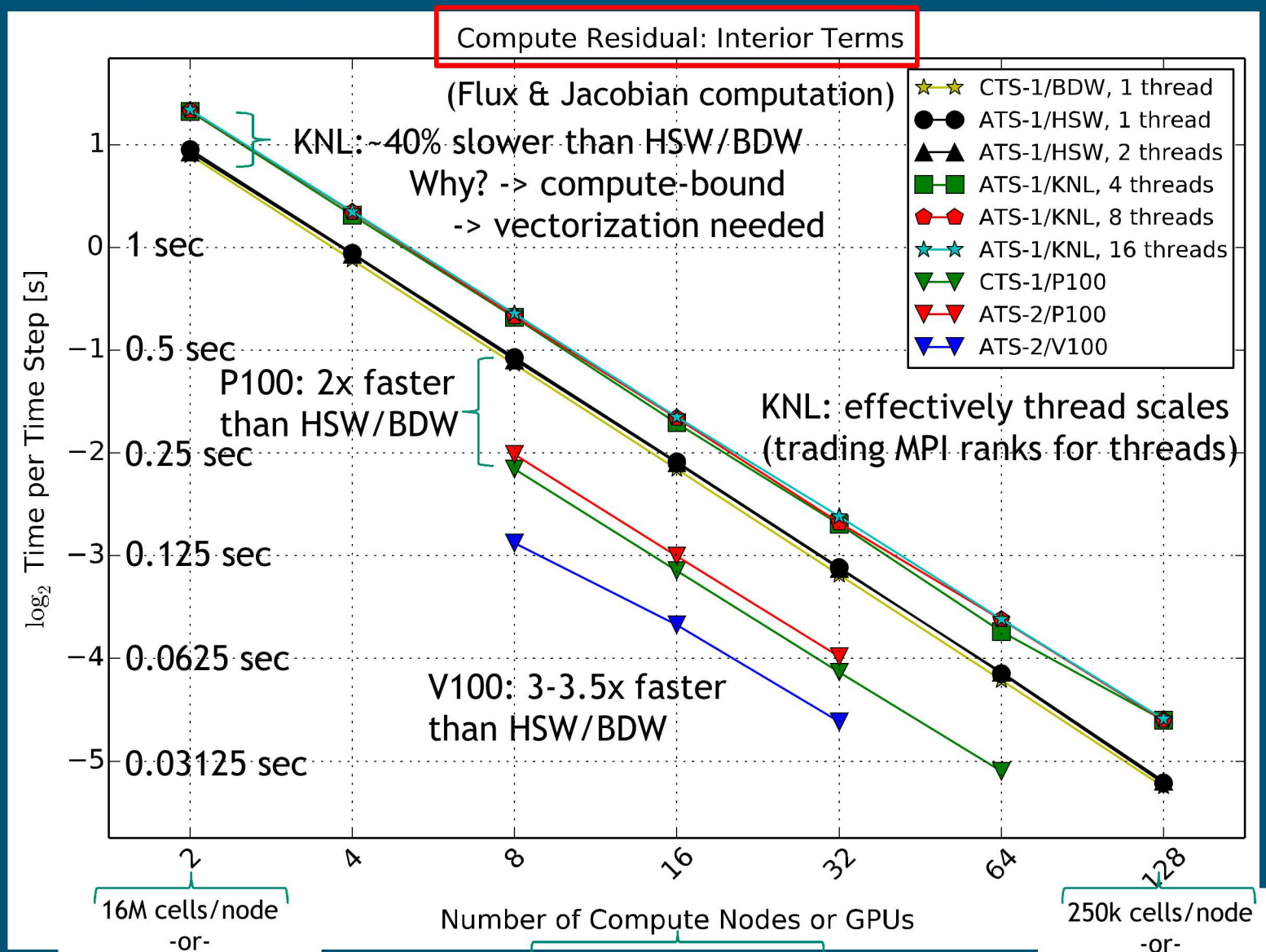
- Intel Xeon systems
  - Haswell nodes – ATS-1/HSW ('advanced technology system', Trinity)
  - Broadwell nodes – CTS-1/BDW ('commodity technology system')
- Intel Xeon Phi
  - Knight's Landing nodes – ATS-1/KNL (Trinity)
- Nvidia GPU
  - Broadwell/Pascal100 nodes – CTS-1/P100 (GPU testbed)
  - Power8/Pascal100 nodes – ATS-2/P100 (Sierra early-access testbed)
  - Power9/Volta100 nodes – ATS-2/V100 (Sierra look-alike testbed)

Case:

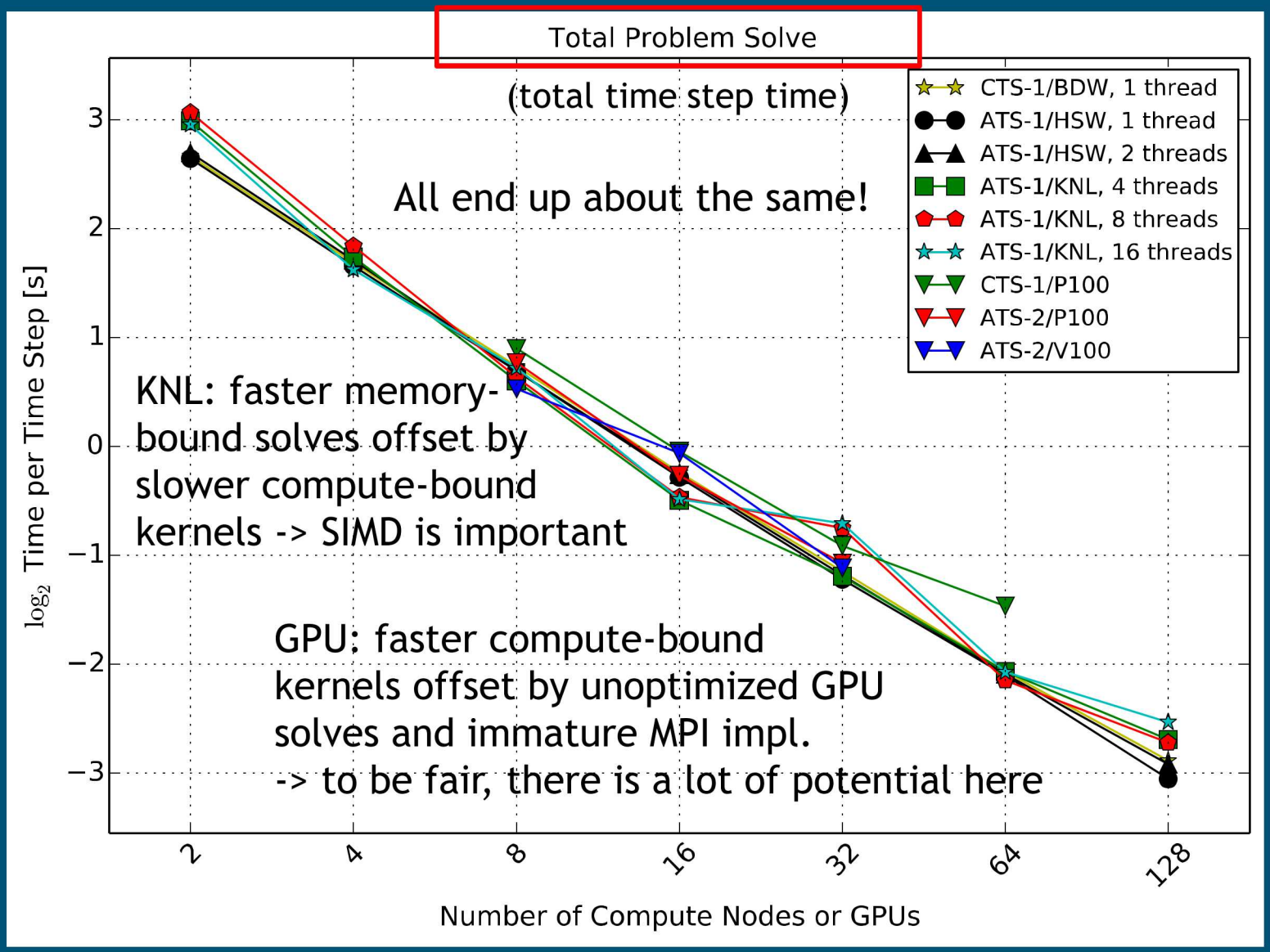
- Perfect gas: ~Mach 6 flow around a sphere-cone geometry
- Reacting gas: ~Mach 9 flow around same geometry with 5-species air model

Strong scaling: 32 M cell grid

PERFECT GAS: STRONG SCALING



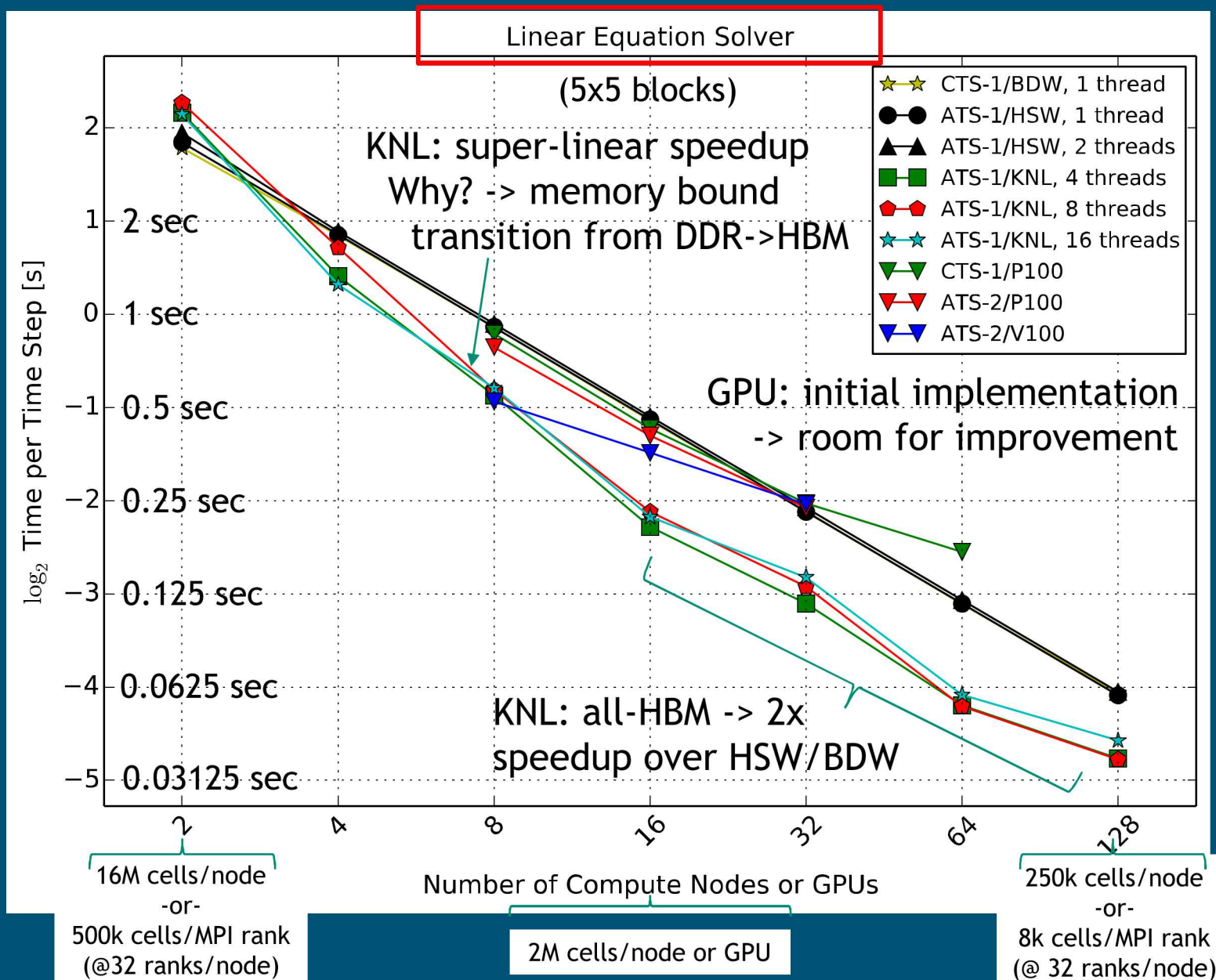
PERFECT GAS: STRONG SCALING (cont'd)



Questions?

PERFECT GAS: STRONG SCALING (cont'd)

log2 scale & lower is faster



PERFECT GAS: STRONG SCALING (cont'd)